SRI International

THE REWRITE RULE MACHINE PROJECT

Final Report
SRI Project ECU 1243

May 1, 1989

By:

Joseph A. Goguen
José Meseguer
Sany Leinwand
Timothy Winkler
Hitoshi Aida

Prepared for:

Office of Naval Research
Information Sciences Division
800 N. Quincy St.
Arlington, VA 22217-5000

Attn: Dr. Richard Lau

Contract No. N00014-85-C-0417

SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
(415) 859-3044

SRI International®
333 Ravenswood Ave. • Menlo Park, CA 94025
415 326-6200 • TWX: 910-373-2046 • Telex: 334-486

The Rewrite Rule Machine Project*

Joseph Goguen†, José Meseguer, Sany Leinwand,
Timothy Winkler† and Hitoshi Aida‡
SRI International, Menlo Park California

Abstract

Current generation parallel machines are typically either coarse-grain or fine-grain. Each design has certain limits, either in the amount of parallelism that it can effectively exploit, or else in the types of problem for which it is suitable. For example, only very homogeneous computations can make efficient use of today's fine-grained machines. However, many complex computations are *locally* homogeneous, but not *globally* homogeneous. The Rewrite Rule Machine (RRM) resolves this dilemma through its *hierarchical architecture*, which has several levels of organization. The lowest level is the cell, which stores a token and pointers to other cells; this organization avoids the von Neumann memory access bottleneck. Next, an ensemble contains many cells, representing a complex data structure to which rewrite rules are applied under the direction of a common controller. A cluster consists of many ensembles which cooperate in larger computations. Ensembles support fine-grained homogeneous parallelism, while clusters support coarse-grained inhomogeneous parallelism. This multi-grained parallelism allows the RRM to exploit the *local homogeneity* that is typical of many complex computations. The RRM can support the efficient execution of many different programming paradigms by compiling them into its concurrent rewriting model of computation. In particular, the RRM can efficiently execute a family of declarative languages that we are developing to extend the advantages of functional programming to object-oriented and logic programming, and to make programming and reprogramming substantially easier than with conventional languages. However, conventional imperative languages, both sequential and parallel, can also be compiled into the concurrent rewriting model. This paper summarizes the RRM architecture, model of computation, and programming techniques, including features of the model of computation that extend ordinary graph rewriting to permit efficient implementation of object-oriented programming. The paper concludes with some simulation results showing that one RRM ensemble has roughly 50 times the power of a Sun-3/60. We consider this very encouraging.

1 Introduction

The goals of the Rewrite Rule Machine (RRM) project are to achieve:

- *efficiency*, through multi-grain massively parallel execution;
- *generality*, supporting both symbolic and numerical computation, as well as both homogeneous and inhomogeneous computation;
- *programmability*, by supporting a wide variety of languages, from the traditional imperative to the modern declarative, including the object-oriented, logic (i.e., relational), and functional paradigms, as well as their combinations; and

*Supported by Office of Naval Research Contracts N00014-85-C-0417 and N00014-86-C-0450, and NSF Grant CCR-8707155.

†Programing Research Group, University of Oxford.
‡On leave from the University of Tokyo.

- ***semantic flexibility and precision,*** by compiling all languages into a common model of computation called **concurrent rewriting.**

Thus, the RRM project aims to develop architectural concepts and a model of computation, to bridge the gap between high level programming and massively parallel execution. The major innovations in architecture are the hierarchical, multi-grain organization, the combination of memory, processing, and communication capabilities at the cell level, the local interpretation (by cells) of SIMD micro-code, and the use of autonomous hardware processes for maintenance tasks such as garbage collection and dynamic data relocation. The model of computation adds extended variables to ordinary graph rewriting in order to obtain efficient implementation of object-oriented programming. The programming languages that we are developing extend the advantages of functional programming to the object-oriented and logic programming paradigms.

1.1 Parallel Architectures

This subsection reviews some basic distinctions in parallel computer architecture, to set the stage for our multi-grain design. The effective use of massive parallelism poses some difficult requirements for both hardware and software, including:

1. assigning tasks to processors that can execute them in parallel,
2. hardware support for efficient inter-processor communication, and
3. effective control of task execution.

Parallel architectures can be characterized by how they try to meet these three requirements:

1. **Granularity** reflects the size of the tasks to be executed.
   - **Fine-grain parallelism** uses very small tasks, often a single instruction, that need to exchange data constantly. Fine-grain architectures typically have many components that cannot do meaningful computations alone, but that can achieve high performance as an aggregate. The efficiency of inter-task communication is critical here.
   - **Coarse-grain parallelism** uses large tasks that rarely need to exchange data. Components of a coarse-grain architecture are capable of independent computation, and the efficiency of inter-task communication is less critical.
2. The options for **inter-processor communication** are closely related to granularity:
   - Processors may have **shared memory.**
   - **Messages** may be sent over a bus or interconnection network.
   - **Direct data exchange** may occur between directly connected processors.

   Shared memory is a resource bottleneck that eventually limits the number of tasks that can be efficiently executed concurrently. Message passing uses explicit access requests, and can be very flexible in matching changing computational needs to available resources. However, buses and interconnection networks are also resource bottlenecks. Thus, message passing and shared memory are most suitable for coarse-grain machines

in which data transfer is sufficiently rare. Direct data exchange, as in pipelined processors and systolic arrays, moves data over prearranged paths, such that whatever data is needed by a particular processor arrives there on time. This can be very efficient, but it imposes rigid limitations that can make programming very difficult, or even impossible.

3. The **control of execution** can be

   - **centralized (SIMD)**, with one controller broadcasting the same instructions to all processors, or
   - **distributed (MIMD)**, where each processor executes its own sequence of instructions on its own data.

   Distributed control is very flexible, but is quite costly because of the need for many independent controllers and for inter-processor synchronization. Centralized control can be very efficient, but it cannot be efficiently applied to inhomogeneous problems.

The goal of massively parallel architectures is to achieve greater power. It is easy to put together a thousand processors and claim a thousand-fold speedup, but it can be very hard to make effective use of such a system. In fact, current generation machines are easy to program in traditional languages, but cannot exploit fully their parallelism except on a narrow class of problems. Thus, the research challenge is to design machines that can efficiently exploit massive parallelism for a wide range of problems, and are also easy to program. The RRM answers this challenge through its **hierarchical, multi-grain** architecture, its **concurrent rewriting** models of computation, and its declarative ultra-high level languages.

1.2 Multi-Grain Parallelism

Experience shows that many computations are *homogeneous*, in the sense that many instances of one instruction can be applied simultaneously at many different places in the data. For example, sorting, searching, matrix inversion, the fast Fourier transform, and arbitrary precision arithmetic all have this character. For such computations, a SIMD architecture seems advantageous.

On the other hand, large, complex computations tend to have many different parts with little or no overlap among their instructions; that is, large, complex computations tend to be *globally inhomogeneous*. SIMD architectures can be very inefficient in such cases. We say that computations that are locally homogeneous but globally inhomogeneous have **multi-grain parallelism**. Architecturally, this suggests a network of many independent fine-grain SIMD processors.

Thus, technological progress has created an opportunity to answer the real need for large, complex computations. Unfortunately, there are serious obstructions to exploiting this opportunity at both the conceptual and the programming language levels. In fact, no well known architecture, model of computation, or programming language is well suited to multi-grain parallel computation. In particular, the von Neumann machines, models of computation, and languages are inadequate, because they are inherently sequential. Similarly, both coarse- and fine- grain architectures can only exploit part of the parallelism

available in multi-grain parallel computations. However, the Rewrite Rule Machine's hierarchical multi-grain architecture, its concurrent rewriting models of computation, and its declarative languages are specifically designed to exploit full multi-grain parallelism.

1.3 Performance

Of course, the test of the ideas summarized above (and described in greater detail below) is whether or not a machine based on them is (relatively) easy to program for high performance on typical inhomogeneous problems. While we have not yet been able to test this directly, we have been able to obtain accurate performance estimates at the ensemble level. As described in Section 5, a single ensemble has roughly 50 times the performance of a Sun-3/60, i.e., about 150 MIPS, for typical homogeneous problems. We consider this to be highly encouraging. The techniques discussed in [11] appear to be quite adequate for inter-ensemble coordination, and thus we expect this performance to scale to inhomogeneous problems where sufficient parallelism is available for exploitation; this is because the volume of data exchanged between ensembles is typically much less than is exchanged within ensembles. Also, the example programs in this paper are quite typical in their simplicity; see [20] for several larger programs.

2 Model of Computation

The **concurrent rewriting** model of computation plays two important roles in the RRM project:

1. It provides an abstract description of what the RRM is supposed to do, and thus a correctness criterion for its architectural design.

2. It serves as a common target language for compilers from a variety of source languages; this allows the second stage of compilation, which produces ensemble controller code, to be shared among all languages.

Issues relating to the first role are further discussed in Section 3, while those relating to the second are discussed in Section 2.3 below. This model of computation adds the concepts of partitioned rewriting (see Section 2.2 below) and extended variables (see Section 2.3 below) to the usual term and graph rewriting models, such as Dactl [5], Rediflow [24], Id [1], and Alice [21]; see [23] for an overview of graph reduction models of computation.

2.1 Data Representation

Although terms, such as $f(x)+g(x)$, can be visualized as trees, it is preferable to represent them as graphs, for the following reasons:

- Storage can be reduced by sharing common data, such as x in $f(x)+g(x)$. Such sharing also reduces the amount of computation required, since a shared subcomputation can be performed just once.

- Replacement is much simpler for graphs, because maintaining a tree representation requires copying (possibly large) structures when variables occur more than once in the righthand side of a matched rule.

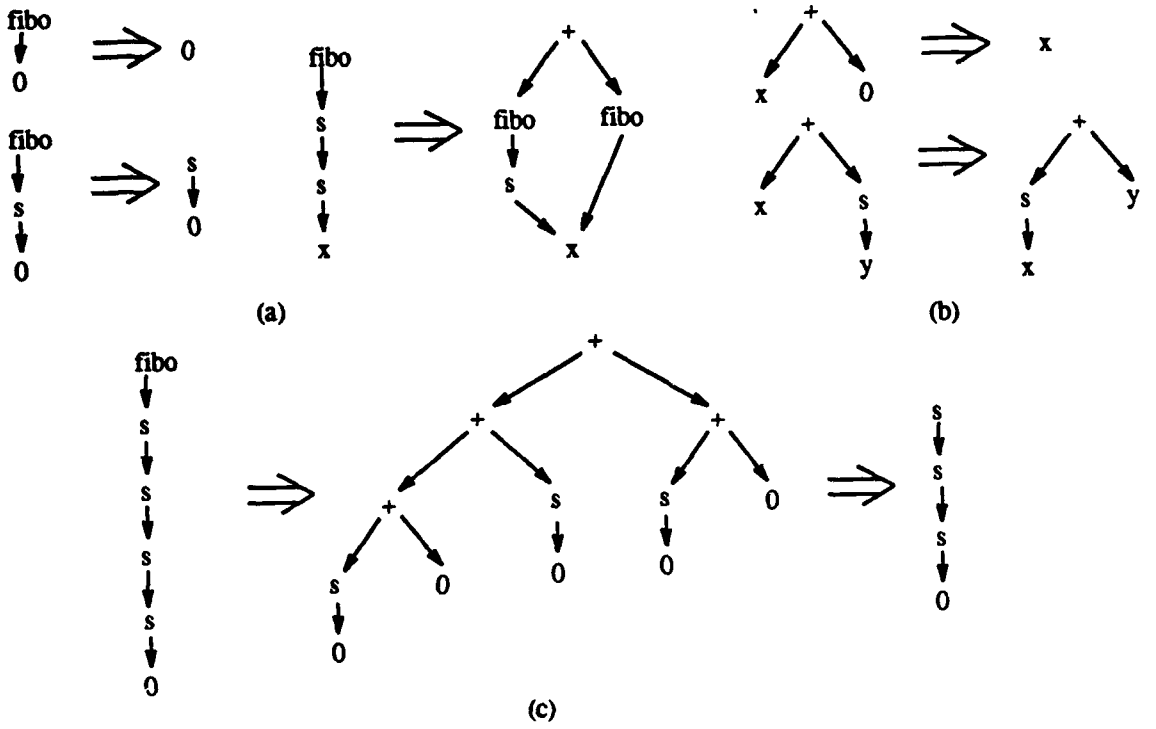

Figure 1: Rewrite Rules for Fibonacci and Addition

Moreover, graph structures are unavoidable for object-oriented programming, because of multiple access to objects [17]. The nodes of these graphs are labelled by operator and constant symbols; constant symbols are considered to be operator symbols with no arguments. In general, type (i.e., sort) restrictions may be attached to operator symbols.

It is easy and natural to represent familiar data structues by such labelled graphs, and although it may seem surprising at first, it is also easy and natural to represent algorithms that manipulate these structures as sets of rewrite rules. Much of this follows from well-known results about implementing abstract data types using term rewriting, as embodied, for example, in the OBJ3 language (see Section 4.2 and [20]).

2.2 Concurrent Rewriting

An RRM computation starts with a graph and a set of rewrite rules. The rules are applied until the graph is **reduced**, in the sense that no rule is applicable[1]. Each rule has a **lefthand** and a **righthand side**, constructed from operator and variable symbols. Variables can be instantiated with any graph (of the appropriate sort), and a set of instantiations for variables is called a **substitution**. Rewriting has two phases, called **matching** and **replacement**. The matching phase finds a substructure of the data graph, called the **redex**, such that some substitution yields the redex when applied to the lefthand side. Then the redex is replaced by the corresponding substitution instance of the righthand side.

Concurrent rewriting allows applying multiple rules at once, at multiple places in the data, and is well adapted to massively parallel computation because no explicit constructs are required to achieve or to describe parallelism. This greatly eases programming. Let us consider a simple example, the Fibonacci function, as defined by the equations

[1]This restriction can be relaxed to implement so-called *perpetual* processes.

```
fibo(0) = 0 .
fibo(s(0)) = s(0) .
fibo(s(s(x)) = fibo(s(x)) + fibo(x) .
```

where the natural numbers are represented using only the constant 0 and the successor function s, i.e., using Peano arithmetic, so that, for example, 3 is represented by s(s(s(0))). Figure 1(a) shows these rules in graphical form; notice the sharing of x in the righthand side of the third rule. Similarly, Figure 1(b) shows the rules for the + function. These rules can be applied to any graph containing fibo, +, s, and 0 symbols. For example, the three graphs in Figure 1(c) show an initial data graph, then the result of applying the rules in Figure 1(a) to it, and finally the result of also applying the rules in 1(b), giving a graph with only 0 and s symbols, i.e., an integer.

Two different special cases of concurrent rewriting are:

1. **Parallel rewriting**, which allows the application of a single rule in multiple places at once. Homogeneous computations are typically handled well by this strategy.

2. **Partitioned parallel rewriting**, which partitions the data into **domains**, such that just one rule at a time is applied at many places within each domain. The domains are locally connected parts of the data structure, and may change dynamically.

Parallel rewriting models fine-grain SIMD parallelism, whereas partitioned parallel rewriting, being locally SIMD, but globally MIMD, models multi-grain parallelism, and extends the effectiveness of parallel rewriting to computations that are only locally homogeneous.

An **evaluation strategy** can be given as an annotation on an operator symbol to impose specific restrictions on the order of rewriting its argument subgraphs. These annotations can be used to improve performance, and also to support the explicit programming of concurrency that is needed for systems programming [11].

A set of rules is **confluent** (sometimes called Church-Rosser) if the order of applying its rules is immaterial to the result. The fact that large confluent subsets of rules are quite typical in practice permits some very significant simplifications of the RRM architecture, for the following reasons:

1. The rules within a confluent set can be scheduled in any order, and any one rule can be applied at several different places in parallel.

2. Given a confluent set of rules, it is not necessary that each match of a given rule to a given substructure is replaced when it is first recognized, provided that the rule will be tried again later. This means that local data access can fail without compromising correctness.

The first property removes the sequential control bottleneck inherent in the von Neumann model of computation, while the second allows both the high performance of fast local connections, and the flexibility of remote connections. However, the RRM is by no means limited to Church-Rosser programs.

2.3 Support for Programming

Compilation for the RRM divides naturally into two phases, the first from the source language into the model of computation, and the second from the model of computation into

```
obj FIBO is
 protecting NAT .
 op fibo : Nat -> Nat .
 var N : Nat .
 cq fibo(N) = N if N < 2 .
 cq fibo(N) = fibo(N - 1) + fibo(N - 2) if 2 <= N .
endo
```

Figure 2: Fibonacci Code in OBJ

Figure 3: The Logical Structure of a Cell

ensemble controller code. This simplifies compiler construction, because most of the hard optimization work can be done by a common backend. Moreover, this backend is not as difficult as for many other machines, because the optimizations that would be most difficult for a compiler are instead done at run-time by autonomous processes within ensembles; see Section 3.3 below. All this makes it easier to implement many different languages on the RRM.

Graph rewriting has been widely suggested for implementing higher-order functional languages; for a very nice survey, see [22]. Ordinary graph rewriting uses ordinary first-order variables, which match complete substructures. Note that these substructures need not be reduced; in fact, imposing this conditions would amount to requiring strict (i.e., eager, or bottom-up) order of evaluation. The RRM project has developed a notion of *extended* graph rewriting that extends the notion of variable instantiation, and greatly facilitates implementing other programming paradigms; see Section 2.3.2.

2.3.1 A Simple Functional Program

We use the simple program for Fibonacci numbers given in Figure 2 to illustrate some basic features of OBJ. The most basic OBJ entity is the **object**, a module encapsulating executable code. The keywords `obj ... endo` delimit the text of an object. Immediately after the initial keyword `obj` comes the object name, in this case `FIBO`; then comes a declaration indicating that the built-in object `NAT` is imported. This is followed by declarations for the new sorts of data (in this case there are none) and the new operations (in this case, `fibo`), with information about the sorts of arguments and results (here, both are `Nat`). Finally, a variable of sort `Nat` is declared, and two equations are given; the keyword `cq` indicates that these are conditional equations (unconditional equations use the keyword `eq`). `<` is the "less than" predicate, and `<=` is the "less than or equal" predicate; these are imported from `NAT` along with the addition and subtraction operations.

Before describing how to implement OBJ on the RRM, we need more information about RRM design. The RRM has been designed *hierarchically*, that is, as a series of models, each more concrete than the one above. The highest levels are actually semantic rather than architectural; for OBJ, these model are equational logic and term rewriting, the former providing a denotational semantics, and the latter an operational semantics. We now discuss the most abstract architectural model for the RRM, the **cell machine**, consisting of an

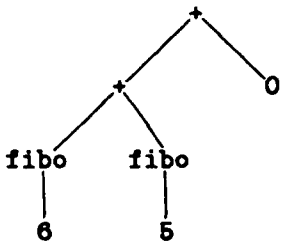

Figure 4: The Tree of a Term

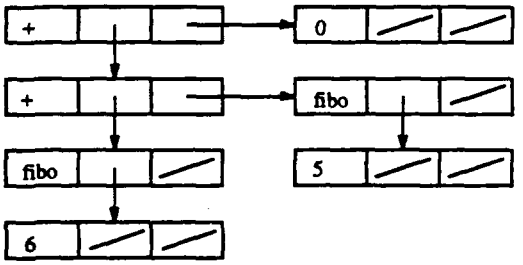

Figure 5: The Cell Representation of a Term

arbitrary number of cells, each with three major registers and an arbitrary number of "flags," which can be "set" or "unset" (i.e., "up" or "down"). The **token** register stores the "content" of a cell, while its **left** and **right pointer** registers each give the location of another cell (or else are empty)[2]. The flags are used to store local status information during matching and rewriting. Figure 3 shows the *logical* structure of a cell; of course, the *physical* structure is more complex, but our subsequent diagrams actually simplify further and omit the flags. This model assumes that each cell can communicate directly with any other cell; Section 3.3 discusses how the actual RRM realizes the same logical power using only local connectivity.

It is evident how to represent a binary tree (or dag) in such a cell machine; for example, Figure 5 shows the tree of Figure 4. We now consider how to implement rewriting with SIMD instruction streams that are broadcast simultaneously to all cells from the central controller. The following are some typical controller instructions: set a certain flag if the token has a certain value; fetch a token (or pointer) from another cell whose location is known; and set the token to a certain value if a certain flag is set. In this model, every instruction is interpreted and (if applicable) executed in each cell using only information that is *local* to that cell.

Although arithmetic for the natural numbers is provided by the RRM hardware, the following discussion will use a basic Peano representation, with constructors the constant 0 and the unary successor operation s, as previously discussed in Section 2.2, and shown in full in Figure 6. Then the rewrite rule

```
fibo(s(s(N))) = fibo(s(N)) + fibo(N)
```

from Figure 6 can be implemented by first identifying each cell that contains the token

[2]Two pointers are sufficient, because an n-ary source level operation symbol can be translated into $n-2$ binary operations for $n > 2$.

```
obj FIBO is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  op fibo : Nat -> Nat .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(M) + N = s(M + N).
  eq fibo(0) = 0 .
  eq fibo(s(0)) = s(0) .
  eq fibo(s(s(N))) = fibo(s(N)) + fibo(N) .
endo
```

Figure 6: Peano Fibonacci Code in OBJ

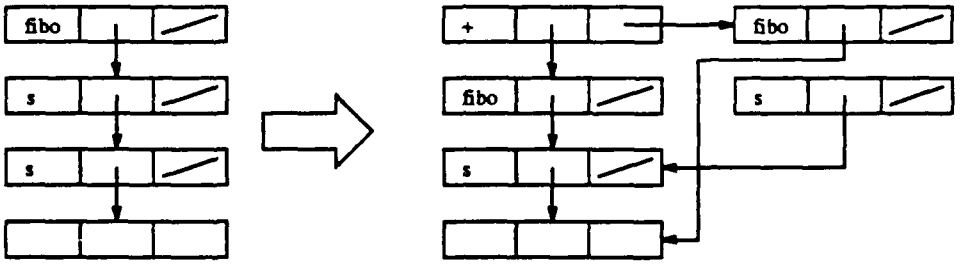

Figure 7: Rewriting a Cell Representation

`fibo`, and then checking that the cell indicated by its left pointer contains a successor that points to another successor (this check could also be done bottom-up).

Once the instances of the pattern `fibo(s(s(x)))` are identified, then replacement can begin; for example, we may replace the token `fibo` at the root of the pattern by `+`, replace its left pointer by a pointer to its `s(x)` cell, and set its right pointer to the `x` cell. See Figure 7. Notice that there is now one less pointer to the first `s` cell, so that it should be collected as garbage if there are no other pointers to it. Also notice that a dag structure has been created from what might previously have been just a tree structure. The following *copy rule* expresses an important restriction on modifying cells during term rewriting:

> *If there is more than one active pointer to a cell, then it cannot be modified, and must instead be copied, unless it is the root of the redex.*

2.3.2 Extended Variables

[9] and [6] show that a certain second order equational logic is a natural extension of standard first order equational logic. It is exciting that there is a corresponding natural extension of term rewriting, called **extended rewriting**, that can be realized on the RRM just as easily and efficiently as standard rewriting. The idea is to allow **extended variables** in the lefthand side of rules. There are two special cases. In the first, the variable can match *partial* data structures, such as the upper part of a tree. In the second, an **operator symbol variable** can match a *single* operator symbol, and its arguments match appropriate

substructures. For example, in the rule

```
m(X(a(A),b(B)),D) = X(a(A + D),b(B + D))
```

the variable X matches operation symbols of appropriate arity, while A,B,D match subtrees as usual, and a,b are unary operation symbols. These two cases do not give the most general forms of second order rewriting, which (as far as we know) cannot in general be efficiently implemented.

The following results are relevant to implementing other language paradigms with this extended graph rewriting:

1. **Object-oriented programming** can be implemented using simple and natural extensions of concurrent rewriting [17,18]; see Section 4.3. This is important because there are many problems where pure functional programming is unnatural or difficult to apply, such as input/output and databases. Logic (i.e., relational) programming can also be handled in a natural way; see Section 4.4.

2. The essential power of **higher-order programming**, including software modularity and reuse, can be achieved *without higher-order functions* [6]. This is important because, in general, higher-order functions can not be implemented as efficiently as first-order functions, and are also require sophisticated strictness analysis for compilation.

3. The RRM architecture implements extended rewriting in a natural and efficient way, thus giving efficient support for non-functional programming paradigms.

4. The extensive work on initial algebra semantics and its applications to programming [3,12,18,14] provides many theoretical results that can be applied to concurrent rewriting [11].

3 Architecture

This section presents some further details of the RRM architecture.

3.1 Levels

The RRM architecture can be described at the following levels of granularity:

1. A **cell** stores a node of a graph.

2. A **tile** provides shared communication and processing resources for a small number of cells in the same ensemble.

3. An **ensemble** includes many cells executing instructions broadcast by a common controller.

4. A **cluster** interconnects many ensembles to cooperate in a larger computation.

5. A **network** interconnects several clusters to give a complete RRM.

Ensembles are designed for efficient fine-grained homogeneous computation in SIMD mode, while clusters are designed for efficient inhomogeneous computation in MIMD mode.

3.2 Cell Architecture

Cell structure is determined by the fact that cells provide physical representations for the abstract nodes in a graph, as well as the basic processing power for rewriting. Previous work [27] has suggested using flags stored in cells to implement the matching of lefthand sides against data. Simple instructions can check the presence of a specific flag or token in a cell. Then complex patterns can be matched by progressively identifying larger and larger substructures, and marking them with further flags. This process can be primarily bottom-up (from the leaves of the pattern to its root), or top-down, or a combination, and it requires that cells have access to a comparator for tokens and flags. Following a successful match, a new structure is built, which in general requires allocating new cells and storing pointers in them. When completed, this new structure replaces the redex by overwriting the root. This suggests that cells should have the following:

- A **token** to represent the node's operator symbol.
- **Pointers** to other structures that serve as arguments to the operator symbol represented by the token.
- **Flags** to summarize the local state of computation at a cell, e.g., that a certain condition has been checked, or that a certain substructure is reduced.
- **Temporary registers** to store pointers into newly created structures.

Partitioning computation among processors keeps the number of distinct operator symbols small enough so that only (say) 8 to 10 bits are needed to represent a token. Pointers can also be fairly small (say 10 bits), because the number of cells in a single ensemble is limited by available silicon area. Two pointers suffice, because operator symbols taking more than two arguments can be represented using several binary operator symbols. We have found that three temporary registers are sufficient for rapid replacement.

It is also important to keep cells as simple as possible, so that several hundred, plus a controller, can fit in an ensemble implemented on a single chip. This will avoid the delays of off-chip communication, and permit high clock rates. Efficient numerical computation on the RRM needs capabilities at the cell level beyond those required for rewriting. Although numerical operations could be implemented from basic principles by representing natural numbers using only successor and zero (i.e., using Peano arithmetic), this would be much too slow. Instead, we can let cells perform simple operations on small (8 to 10 bit) numbers, including signed addition, negation, shift, and bit operations. The incremental cost over the already required equality comparison on tokens is small. Then RRM compilers can implement a complete set of arithmetic operations from these built-in cell operations. A redundant representation of arbitrary precision numbers as trees of small integers [29] permits highly parallel arithmetic operations, and can very effectiently exploit the RRM.

3.3 Ensemble Architecture

An ensemble consists of many cells, a shared controller that broadcasts instructions to them, and local storage for the rules that are currently applicable, all on a single chip. Of course, cells should be interconnected with a regular mesh of fixed degree (e.g., rectangular or hexagonal), to effectively use silicon floorspace. The following describes some solutions to problems at the ensemble level.

3.3.1 Local Interpretation

The instructions broadcast by the controller are *interpreted in the local context of each cell.* For example, an instruction might ask each active cell to access the cell pointed at by one of its registers. This means that compiled code does not need to take account of the actual allocation of nodes to cells, which greatly simplifies both the control of computation and the effective utilization of communication resources.

3.3.2 Ensemble Resource Management

The replacement phase of a rewrite constructs the righthand side of a rule, and may fail if there are not enough free cells or if communication is blocked. Therefore, it must be possible to abort replacement at any point without corrupting the original structure. Several measures are taken for this purpose:

- The new structure is built entirely from newly allocated cells.
- If the righthand side cannot be successfully built, because of allocation or communication failure, then the partially constructed righthand side is deallocated and the redex is left unchanged.
- Once the righthand side has been constructed, the redex is changed by an atomic `commit` instruction that is guaranteed to succeed eventually.

Whenever a cell uses a pointer to a remote cell, the current instruction is aborted for that cell (other cells may continue) and a relocation request is issued, which will eventually cause the remote cell to be relocated to a physically adjacent cell, as described in more detail below; however, if the cells involved in the operation are all local, then the operation is performed immediately.

To make better use of resources, the silicon area of an ensemble is divided into a regular array of **tiles**, each of which provides communication capabilities and some shared operations for a small number of cells; 8 is the number currently under evaluation. A port connecting a pair of adjacent tiles is used for all inter-cell communication between that pair. If there are multiple requests for a port, one request succeeds and the others fail. Simulation studies show that this arrangement is quite efficient provided the occupancy rate of cells is below 50%.

3.3.3 Dynamic Data Representation

The RRM avoids the overhead of message passing and the inflexibility of data passing by using what we call **dynamic data representation**. In this approach, adjacent cells are directly connected by short high-speed wires, so that matching can be very efficient when all cells are physically adjacent. When replacement requires new cells, adjacent free cells can usually be found, but if all adjacent cells are full, then cells must be relocated. Pointers to remote cells can also arise when a newly built structure uses a substructure obtained by matching a variable, or when a remote cell is relocated without relocating its descendents.

3.3.4 Autonomous Cell Relocation

Since remote connections cannot be used for direct data transfer, we propose to use **autonomous hardware processes** in which cells that are not currently allocated may be used

to relay messages and to copy cells. Any request for data transfer with a remote cell fails, and then starts an autonomous process that will eventually relocate the target cell to a cell that is physically connected to the requester. This is achieved by creating an autonomous message cell that "moves" toward the target cell by allocating another cell closer to the target, copying its state into it, and then deallocating itself. When the target cell is reached, it is copied back to the requester by a similar process, but more slowly, since a cell contains more data than a message, and several instructions may be required for transfer. This delay causes little difficulty, because the controller's code cycles also tend to be rather long. Alternatively, multiple independent messages could be used.

3.3.5 Global Feedback

Correct implementation of concurrent rewriting on the RRM requires knowing when to stop applying rules, that is, knowing when a subgraph is reduced. If there are no non-constructor operator symbols in a graph, then it must be reduced. We propose to test this condition using a global feedback network, from the array of all cells in an ensemble to their controller. This might be a binary tree, which computes the OR of incoming signals at each node. The controller could then broadcast a command asking whether any cell contains a certain token (or more generally, a certain combination of token and flags). The anwer would then be known after some delay. This network could also simplify and speed up certain other tasks.

3.4 Cluster Architecture

A cluster contains many ensembles, a large backup memory, and a connection to a conventional computer that stores the complete set of rewrite rules. Communication protocols, rule distribution strategies, and load distribution are important issues at this architectural level. Clusters are relatively independent entities that need to communicate relatively rarely. Thus they admit coarse-grain parallelism and support multi-grain parallelism.

3.4.1 Data Partitioning

Inter-ensemble communication can be reduced by properly partitioning data among ensembles. Theoretical work on the partitioned parallel rewriting model suggests that there are natural "articulation points" to guide this process [11]. Also, computations that grow too large for one ensemble can be distributed over several ensembles. Some computations require an exponentially increasing number of cells, and hence can overwhelm any available resources. Rather than distribute computations into ensembles outside their original cluster, excess data could be stored in the local cluster's backup memory until some local ensembles become less loaded.

3.4.2 Program Partitioning

Ensembles are designed for fine-grain homogeneous parallel computation, and are not efficient for inhomogeneous computations. Data partitioning ensures that the data in each ensemble is fairly homogeneous, while program partitioning ensures that each ensemble contains the rewrite rules appropriate to its data. The rule sets are distributed from a conventional computer. Since a cluster contains many ensembles, each with its own rule set, this conventional computer could become a bottleneck. However, the local storage for rules in ensembles acts as a cache, and should eliminate this problem.

3.4.3 Cluster Synchronization

Ensembles are relatively independent entities, but when a data structure extends over several ensembles, they must cooperate in rewriting [11]. Unfortunately, the matching algorithm is more complex in this case. The difficulties are that inter-ensemble communication is very slow compared with intra-ensemble communication, and that since each ensemble is independently and asynchronously controlled, protocols for determining whether a match is successful will necessarily be relatively slow. However, there are reasons to believe that such synchronizations will be needed rarely enough so that there will not be a large negative impact on performance [11].

3.5 Network Architecture

A complete RRM is a network of relatively few clusters, used for solving multiple or very large problems. A general purpose interconnection switch is appropriate at this level. Tokens must have long global names, because the 8 to 10 bits used inside a cluster are not sufficient to identify all the operator symbols that may occur in a larger program. Therefore translation is required when two clusters interact. Since inter-cluster communication is slow compared to intra-cluster communication, rewriting is not allowed to split across clusters. A conventional computer provides a user interface to the RRM as a whole.

3.6 Novel Architectural Concepts

This subsection summarizes some of the unusual architectural concepts that have been developed by the RRM project.

1. **Combine processing, storage, and communication** at the cell (and tile) levels, to avoid the memory access bottleneck of von Neumann architectures and the memory latency problem of dataflow architectures.

2. **Locally interpreted code.** The instructions broadcast to cells are interpreted locally, by each cell in its own context. This differs significantly from traditional SIMD design, in which the central controller uses the ***physical location*** of data in its instructions. For example, a traditional SIMD instruction might request each active cell to access the cell physically above it. But an RRM ensemble controller instruction can request that each cell access the cell pointed at by one of its registers. Local instruction interpretation allows much greater flexibility, which greatly simplifies compilation, and also improves efficiency, because requests for scarce resources are resolved in the specific context of each cell, rather than by the controller.

3. **Fast local connections.** Connect physically adjacent cells in an ensemble by short, high speed wires. This allows very fast operations when all the cells involved are adjacent.

4. **Non-critical scheduling.** Since the model of computation requires that rules are executed until the data is fully reduced, the rules must be executed repeatedly; but for many large subsets of rules, the order is arbitrary. This permits a rewrite to be abandoned whenever convenient, provided its redex is left in a consistent state. This in turn allows the RRM to make use of its extremely fast local connections.

5. **Autonomous processes.** Any cell that is either inactive or free during a particular instruction can use its resources for other worthwhile tasks, without hindering normal rewriting. These tasks include garbage collection and cell relocation. The result is a self-organizing array of cells that either interpret the instruction stream, or else reorganize themselves to make better use of available resources.

6. **Dynamic data relocation.** A cell's request for access to data in another cell will only succeed if the other cell is adjacent. This can be accommodated by (autonomously) relocating remote cells to adjacent locations on an as-needed basis. Data sharing is thus achieved by dynamically allocating resources to cells. For example, cells wishing to modify an object will take turns being adjacent to the object.

7. **Design for average load.** Communication is a critical resource in any massively parallel machine. In most cases, a rewrite can be abandoned if it will be tried again later, so communication overload can be handled by some communication requests failing, and then deactivating their originators. This allows communication resources to be designed for average load, rather than for the worst case.

8. **Multi-grain execution.** The RRM extends the efficiency of fine-grain execution to computations that are only locally homogeneous, by dynamically partitioning data into homogeneous domains, each of which is efficiently processed inside one ensemble.

4 Programming

Programming is often the major obstacle to the effective use of a massively parallel machine. There are two basic approaches to this problem:

1. Reuse old programs, often written in old sequential imperative languages.

2. Write new programs in new parallel imperative languages.

The first approach would have considerable merit if it were possible for a compiler to extract sufficient parallelism from "dusty decks." The current state of the art can extract moderate parallelism for many homogeneous computations, often with some reprogramming by hand. In general, the second approach can achieve much greater parallelism, but may yield programs that are difficult to debug, modify, and port to other machines, because of the notorious difficulty of understanding parallel programs, and often also because of machine dependency in the programming language and/or the program. As explained below, these problems are alleviated by our well-defined model of computation and the technology that we are developing to compile declarative languages into it.

4.1 Declarative Programming

While we believe that the RRM can effectively support programs written in traditional languages, we also believe that declarative programming languages can exploit the RRM with greater efficiency, and at the same time offer significant advantages in programming ease and maintenance. Programs in declarative languages tend to describe problems, rather than solutions. From the hardware viewpoint, declarative languages do not prescribe specific orders of execution, and thus provide maximal opportunity for compilers and runtime

systems to exploit parallelism. From the software viewpoint, declarative languages avoid the need to explicitly program parallel tasks, which can be very difficult. Moreover, programs written in declarative languages do not need to be rewritten if the underlying hardware is slightly changed, e.g., if more processors or memories are added, since they are already independent of any assumptions about the underlying hardware. Also, they have simple syntax and semantics, and thus are relatively easy to learn and to compile. Furthermore, the languages that we are developing for the RRM have features to support all phases of program development, from specification and design to maintenance.

This section describes RRM compiler techniques for the three major emerging programming paradigms, the functional, object-oriented, and logic, as well as their combinations. We describe these techniques in terms of languages that we have already developed for the RRM project, but it should be emphasized that these techniques apply to other languages in these paradigms. The techniques are also applicable to more traditional languages. In particular, imperative programming can be seen as a degenerate case of object-oriented programming.

Both functional and (many forms of) object-oriented programming are already quite close to the concurrent rewriting model of computation. Indeed, this model of computation provides a very natural and direct way of expressing the parallel execution of such languages. For logic programming, the correspondence is less direct, and some additional steps are needed. However, these steps are conceptually simple, and also take advantage of parallelism.

4.2 Functional Programming

OBJ [19,3,4] is a functional programming language whose operational semantics is based upon initial algebra semantics. It is well known that this semantics is correctly implemented by rewriting trees (or directed acyclic graphs) under certain simple assumptions. This was first proved in [7], and [11] shows that concurrent rewriting is also correct under the same assumptions. OBJ has no explicit constructs for creating or synchronizing parallel processes. Rather, the parallelism of an OBJ program is *inherent* in the program itself.

OBJ was designed to directly embody various modern software engineering techniques, rather than to provide them indirectly in an environment having separate conventions and notations. These features include:

1. **Parameterized programming**, to support software reuse and wide spectrum integration of design, documentation, rapid prototyping, and specification, with
   - powerful "tunable" **generic modules** that go far beyond Ada's generics or mere functional composition, and are powerful enough to give the power of higher-order programming without its difficulties in understandability and verification [6],
   - **theories**, which provide non-executable axioms to describe semantic properties of modules and module interfaces, as well as their syntax.
   - **views**, which assert semantic properties of modules, and also bind actual modules to formal modules for instantiating generic modules,
   - **module expressions**, which support programming-in-the-large, by describing how to build complex subsystems from previously defined modules, and then actually build them when evaluated,

- **module hierarchies**, whereby old modules may be imported into new modules, and
- **user-definable abstract data types**, not limited to constructors, as they are in most other functional languages.

2. **Subsorts**, which support multiple inheritance, exception handling, partial functions, and operator overloading in an elegant way.

3. **Matching modulo equations**, including the associative, commutative, and identity laws. This greatly increases the power of matching and hence the expressiveness of the language.

4. **Evaluation strategies**, which avoid enslavement to any fixed evaluation strategy, such as eager or lazy, and thus allow greater efficiency in both time and space. Note that strictness analysis is much easier than for higher order languages, and is automatically provided by OBJ3 when users do not give explicit strategies.

OBJ has been rather extensively studied from both the theoretical and practical viewpoints [15,16,19,3,10,4], and there are now several implementations, including OBJ3 at SRI [20], one from Washington State University, three from Great Britain, one from Italy, and one from Japan. One of the British versions has some currency as a commercial product, and over 60 OBJ3 systems have been shipped from SRI at this writing.

4.2.1 Implementation on the RRM

OBJ module expressions are evaluated at compile time to yield a corresponding "flat" set of rewrite rules. These rules are then compiled into instructions to be broadcast within ensembles. Details concerning the instructions and their execution may be found in [25]. This is relatively straightforward, although conditional rules present some problems, and we have not yet studied rewriting modulo various equations in detail.

4.3 Object-Oriented Programming

FOOPS [18] was designed to be a simple, yet expressive and efficient general purpose object-oriented language that also embodies the modern software engineering techniques developed for OBJ. Indeed, FOOPS demonstrates that object-oriented programming can be seen as a natural extension of OBJ-style functional programming, rather than a form of imperative programming. We consider that the most important attribute of object-oriented programming is that the only operations that change memory are **methods**, which update only the local attributes of objects. Notice that inheritance arises automatically for us, from the order-sorted logic upon which even OBJ is based.

In FOOPS, objects, abstract data types, methods, and attributes are all defined in a declarative, functional style. This gives FOOPS a simple syntax and semantics which makes it comparatively easy to read, write and learn. FOOPS is also relatively easy to reason about, since it is based on a formal logical system; indeed, [18] gives what seems to be the first ever rigorous semantics for an object-oriented programming language. OBJ is a proper sublanguage of FOOPS, used to define the abstract data types that provide values for attributes. In addition, FOOPS has declarations for classes, attributes and methods.

4.3.1 Implementation on the RRM

We now illustrate the implementation of FOOPS, using (extended) rewrite rules, with a simple bank account example. A complete development of this example is given in [17]. For our present purpose, it is enough to mention that objects are bank accounts that belong to a class called *Acct*, and that their visible properties are given by two attributes, `bal`, the balance, and `hist`, the history of previous transactions, which is represented as a list of money amounts, with positive or negative sign.

An object, such as `Johnson-Acct` is internally represented as a term, like

```
Johnson-Acct(bal:(500),hist:(200 -100 300 -100 200))
```

where the current values of the balance and history attributes are internally represented by the corresponding arguments of `bal:` and `hist:`. To obtain the attributes of an account, we use the rules

```
bal(X(bal:(M),hist:(H))) = M
hist(X(bal:(M),hist:(H))) = H
```

which, for example, give `bal(Johnson-Acct) = 500` for the account described above. Notice that these rules involve an extended variable `X` that ranges over the operator symbols that represent the names of the objects in the class *Acct*.

Method application is only slightly more complex. The key idea is that the several axioms associated with the method are described by a single rule that produces all the changes at once. For the *credit* method, this rule is

```
credit(X(bal:(B),hist:(L)),M) = X(bal:(B + M),hist:(app(L,M)))
```

which increases the balance by `M` and appends `M` to the account's history.

The RRM architecture supports extended rewriting just as efficiently as it does ordinary first-order rewriting. Thus, it is straightforward and efficient to implement object-oriented programming on the RRM. Dactl [5] adds an assignment feature to graph rewriting to help implement object-oriented programming. The following points summarize the differences between implementing objects and implementing values.

1. **Objects persist**, and can be destroyed only by application of a `delete` instruction.
2. **Objects are locked for method application**, to ensure object integrity, allowing only one match attempt to succeed when several instances of a method refer to the same object. (Notice that this problem cannot arise when instances of different methods refer to the same object, because the RRM is locally SIMD.)
3. **Copying of objects is forbidden**, to ensure uniqueness.

This last point indicates a way in which object-oriented computation actually utilizes concurrent rewriting more efficiently than pure functional computation can.

4.4 Logic Programming

It is widely recognized that the relational paradigm is especially suitable for problems that involve deduction and/or search; typical application areas are natural language processing and expert systems. Since pure Horn clause logic is not powerful enough to support truly

practical programming, the RRM project has chosen to investigate more powerful logics, rather than to graft extralogical features into Horn clause syntax. The results of our explorations include designs for Eqlog and FOOPlog, and some ideas on how to implement them, as discussed below.

Eqlog combines the functional and relational programming paradigms and provides the same parameterization and wide spectrum capabilities as OBJ and FOOPS. Like these languages, Eqlog is based on a rigorous order-sorted logic that provides multiple inheritance, and has a precise initial model semantics. Like FOOPS, Eqlog is a proper extension of OBJ. However, instead of adding classes, methods, and so on, Eqlog adds only one basic thing to the syntax of OBJ, namely **predicates**. To achieve semantic consistency, equality is now regarded as a rather special predicate that is always interpreted in models as *actual identity*. The logic for this is quite well known; it is Horn clause logic *with equality*, and there are rules of deduction that give rise to completeness and initiality theorems [14].

The operational semantics of Eqlog divides naturally into two algorithms, one for solving systems of equations and the other for searching. The first algorithm generalizes standard, syntactic unification, the extreme case being so-called **universal** or **semantic unification**, while the second differs little from the usual Prolog-style implementation of search used in SLD-resolution, except that it exploits the opportunities for parallelism which the RRM allows. These two computational tasks are informally discussed below, and rewrite rules for them are given in [17].

4.4.1 Unification

Unification and rewriting are closely related; in particular, the matching phase of rewriting is a special case of unification. What may be more surprising is that unification can be implemented by rewriting in a way that naturally exploits parallelism. As in the Martelli-Montanari unification algorithm [26], we can represent both unification problems and their solutions as sets of equations. Our approach is to give explicit rewrite rules that transform the former into the latter, and the solutions are then the reduced forms under these rules. (Herbrand's original work on unification can also be seen as an algorithm of this kind.) This allows the RRM to perform unification with no extra architectural support. Moreover, the rules are applied in parallel. See [17] for more detail. This can be seen as a special case of implementing constraint logic programming by constraint propagation rules; other cases can exploit RRM capabilities in a similar manner.

4.4.2 Search

A search for solutions to a query might well explore many parts of the search space in parallel. Given a particular program, solving a query Q can be conceptualized functionally rather than non-deterministically. To find the first n solutions of Q, we reduce the term *show* Q *upto* n to a set of substitutions, represented as a disjunction $\theta_1 \vee \theta_2 \vee ... \vee \theta_n$. One approach to implementing search on the RRM is based on ideas similar to those of J.A. Robinson [28] and K. Berkling [2], but our context is broader, since it involves Horn clause logic with equality, and our functional basis is equational logic rather than the lambda calculus. See [17] for more detail.

4.4.3 Multi-Paradigm Programming

There is not space here for more than a few remarks about FOOPlog [18], which combines all three major emerging programming paradigms, the functional, object-oriented, and relational. It appears that techniques similar to those described above will support reasonably efficient implementation of FOOPlog on the RRM. Moreover, we believe that FOOPlog is an especially suitable language for knowledge processing [8], and in particular, for natural language processing [18,13]. A FOOPlog implementation is in progress at the University of New Hampshire.

5 Performance Estimates

In order to determine whether we are wasting our time on this ambitious and unusual project, it is important to run standard benchmark programs and compare their performance with standard von Neumann processors. It is not easy to compare machines with radically different architectures and models of computation. An ideal comparison with the von Neumann paradigm would use equal clock speed, equal silicon floorspace, equal design effort, equal compiler technology, and equal power in supporting hardware (e.g., caching and memory). We chose a SUN-3/60 for comparison, just because it was readily available. Its clock rate is 20MHz, and we can approximate its 68020 floorspace simply by not considering simulations that involve excessively large graphs, e.g., over a few hundred cells. It is clear that this does not give parity in total silicon area, in design effort, or in compiler technology. We have put perhaps 2 or 3 man-years into the RRM project, as compared with thousands for the von Neumann tradition represented by the SUN-3/60, and we have really very little idea of what techniques could be used to improve RRM performance, whereas many such techniques are already used in the 68020 and its associated hardware and software. Furthermore, a Sun-3/60 has many auxiliary chips for coprocessors, onboard memory, etc. Thus, our estimates are conservative.

Our benchmark problems were to compute the Fibonacci numbers with Peano arithmetic and with machine arithmetic, and to bubblesort lists of numbers. Handcoded programs were run on an RRM simulator which yields quite accurate timing results, using a 12-by-12 grid of tiles of 8 cells, limited to 50% occupancy in each tile. We tried to approximate equality in compiler power by comparing our RRM Fibonacci code with compiled Common Lisp (KCL) code for the Sun-3/60, and our bubblesort program with a good sorting program in the well established low level language C; this is unfair to the RRM because we are still in an early stage of ensemble controller code technology. (However, the handcoded Fibonacci code is essentially the same as produced by our OBJ to RRM compiler.) We then approximated the timing data by simple functions, as shown below:

program	time	comment
RRM Peano fibo	$1.84n^2$	?
Sun Peano fibo	$e^{.8268n} + 1000$	$e^{.8268}$ is about 2.29
RRM arith fibo	$4.8n - 2$	
Sun arith fibo	$e^{.6n} + 350$	$e^{.6}$ is about 1.82
RRM bubblesort	$5.44n$	?
Sun bubblesort	$2.48n^2$	

Times are in micro-seconds. Entries marked "?" may be less accurate approximations. The Peano Fibonacci computation was done for values up to $n = 10$, the arithmatic Fibonacci for values up to $n = 15$, and the bubblesort for lists of length up to $n = 128$. This data indicates that the RRM Fibonacci programs have exponential speedup, while the RRM bubblesort is linear and the SUN bubblesort quadratic, giving a linear speedup.

The speedup factor for the largest common case of the Peano Fibonacci computation run on the RRM simulator is 30, while for the arithmetic Fibonacci it is about 70, and for the bubblesort about 60. This supports a conservative claim that a single RRM ensemble is roughly 50 times faster than a SUN-3/60. This is equivalent to about 150 MIPS, assuming a SUN-3/60 is rated at 3 MIPS.

Of course, the RRM project is still at a relatively early stage, and we have not explored an especially wide range of problems, nor have we tried especially hard to get the best programs, or to test them over a wide range of data. On the other hand, the results seem relatively consistent and our simulator is quite accurate for this kind of timing data, so there is no reason to suppose that more work will have any effect other than to increase our performance estimates at this level. Given these conditions, it seems fair to say that the RRM ensemble's observed factor of roughly 50 over the SUN-3/60 is quite impressive. Moreover, we believe that this performance will scale approximately linearly for larger problems in which parallelism is actually available, and that, especially for inhomogeneous problems, this is much better than can be expected from current generation parallel machines. However, this extrapolation is on less solid ground, since our simulations have not yet been extended to the cluster level.

6 Summary

The following summarize points on which we believe the RRM project has made significant progress:

1. Design and simulation of an architecture based on **hierarchical multi-grain massive parallelism**.

2. Exploration of **autonomous hardware processes** that use idle processing power for maintenance tasks, such as data relocation and garbage collection.

3. The use of **dynamic data representation** and **autonomous processes** to exploit very fast local connections that are not always available.

4. Exploration of **concurrent rewriting** models of computation, to support the design of massively parallel machines, and compilation of a variety of languages onto them, particularly object-oriented languages. This includes extending graph rewriting with second order variables and with persistent subgraphs.

5. Techniques for compiling a wide variety of programming languages onto the Rewrite Rule Machine, and validation of our hypothesis that **declarative multi-paradigm languages** can greatly facilitate programming and reprogramming.

The interplay between concurrent rewriting models of computation, hierarchical multi-grain architectures, and autonomous hardware processes seems to open promising territories for further exploration.

References

[1] Arvind, Rishiyur Nikhil, and Keshav Pingali. I-structures: data structures for parallel computing. In Joseph Fasel and Robert Keller, editors, *Graph Reduction*, pages 337–369, Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 279.

[2] Klaus Berkling. *Epsilon-Reduction: Another View of Unification.* Technical Report, Syracuse University, 1986.

[3] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings, 12th ACM Symposium on Principles of Programming Languages*, pages 52–66, Association for Computing Machinery, 1985.

[4] Kokichi Futatsugi, Joseph Goguen, José Meseguer, and Koji Okada. Parameterized programming in OBJ2. In Robert Balzer, editor, *Proceedings, Ninth International Conference on Software Engineering*, pages 51–60, IEEE Computer Society Press, March 1987.

[5] J.R.W. Glauert, K. Hammond, J.R. Kennaway, G.A. Papadopoulos, and M.R. Sleep. *DACTL: Some Introductory Papers.* Technical Report SYS-C88-08, School of Information Systems, University of East Anglia, 1988.

[6] Joseph Goguen. Higher-order functions considered unnecessary for higher-order programming. In David Turner, editor, *Proceedings, Univeristy of Texas Year of Programming, Institute on Declarative Programming*, Addison-Wesley, to appear 1989. Preliminary version in SRI Technical Report SRI-CSL-88-1, January 1988.

[7] Joseph Goguen. How to prove algebraic inductive hypotheses without induction: with applications to the correctness of data type representations. In Wolfgang Bibel and Robert Kowalski, editors, *Proceedings, Fifth Conference on Automated Deduction*, pages 356–373, Springer-Verlag, 1980. Lecture Notes in Computer Science, Volume 87.

[8] Joseph Goguen. Modular algebraic specification of some basic geometrical constructions. *Artificial Intelligence*, 123–153, 1988. Special Issue on Computational Geometry, edited by Deepak Kapur and Joseph Mundy; also, Report Number CSLI-87-87, Center for the Study of Language and Information at Stanford University, March 1987.

[9] Joseph Goguen. OBJ as a theorem prover, with application to hardware verification. In V.P. Subramanyan and Graham Birtwhistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 218–267, Springer-Verlag, 1989. Also, Technical Report SRI-CSL-88-4R2, SRI International, Computer Science Lab, August 1988.

[10] Joseph Goguen. Parameterized programming. *Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[11] Joseph Goguen, Claude Kirchner, and José Meseguer. Concurrent term rewriting as a model of computation. In Robert Keller and Joseph Fasel, editors, *Proceedings, Graph Reduction Workshop*, pages 53–93, Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 279.

[12] Joseph Goguen and José Meseguer. Eqlog: equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363, Prentice-Hall, 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179-210, September 1984.

[13] Joseph Goguen and José Meseguer. Logical programming for situation semantics. In Mark Gawron, David Israel, and José Meseguer, editors, *Semantics of Natural and Computer Languages*, MIT Press, 1989. To appear.

[14] Joseph Goguen and José Meseguer. Models and equality for logical programming. In Hartmut Ehrig, Giorgio Levi, Robert Kowalski, and Ugo Montanari, editors, *Proceedings, 1987 TAPSOFT*, pages 1–22, Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 250.

[15] Joseph Goguen and José Meseguer. *Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations.* Technical Report to appear, SRI International, Computer Science Lab, 1989. Given as lecture at Seminar on Types, Carnegie-Mellon University, June 1983.

[16] Joseph Goguen and José Meseguer. Order-sorted algebra solves the constructor selector, multiple representation and coercion problems. In *Proceedings, Second Symposium on Logic in Computer Science*, pages 18–29, IEEE Computer Society Press, 1987. Also Technical Report CSLI-87-92, Center for the Study of Language and Information, Stanford University, March 1987.

[17] Joseph Goguen and José Meseguer. Software for the rewrite rule machine. In *Proceedings, International Conference on Fifth Generation Computer Systems 1988*, pages 628–637, Institute for New Generation Computer Technology (ICOT), 1988.

[18] Joseph Goguen and José Meseguer. Unifying object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477, MIT Press, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153-162, October 1986.

[19] Joseph Goguen and Joseph Tardo. An introduction to OBJ: a language for writing and testing software specifications. In Marvin Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189, IEEE Press, 1979. Reprinted in *Software Specification Techniques*, Nehan Gehani and Andrew McGettrick, editors, Addison-Wesley, 1985, pages 391-420.

[20] Joseph Goguen and Timothy Winkler. *Introducing OBJ3.* Technical Report SRI-CSL-88-9, SRI International, Computer Science Lab, August 1988.

[21] Peter G. Harrison and Michael Reeve. The parallel graph reduction machine, ALICE. In Joseph Fasel and Robert Keller, editors, *Graph Reduction*, pages 181–202, Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 279.

[22] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[23] Robert Keller and Joseph Fasel, editors. *Proceedings, Graph Reduction Workshop.* Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 279.

[24] Robert Keller, Jon Slater, and Kevin Likes. Overview of Rediflow II development. In Joseph Fasel and Robert Keller, editors, *Graph Reduction*, pages 203–214, Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 279.

[25] Sany Leinwand, Joseph Goguen, and Timothy Winkler. Cell and ensemble architecture of the rewrite rule machine. In *Proceedings, International Conference on Fifth Generation Computer Systems 1988*, pages 869–878, Institute for New Generation Computer Technology (ICOT), 1988.

[26] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[27] Ugo Montanari and Joseph Goguen. *An Abstract Machine for Fast Parallel Matching of Linear Patterns.* Technical Report SRI-CSL-87-3, Computer Science Lab, SRI International, May 1987.

[28] J. Alan Robinson. *A 'Fifth Generation' Programming System Based on a Highly Parallel Reduction Machine.* Technical Report, School of Computer and Information Science, Syracuse University, 1984.

[29] Timothy Winkler. *Numerical Computation on the RRM.* Technical Report, SRI International, Computer Science Lab, November 1988. Technical Note SRI-CSL-TN88-3.